A Cloud-based Time-Dependent Routing Service

Kalliopi Giannakopoulou[1,2], Spyros Kontogiannis[2,3], Georgia Papastavrou[2,3], and Christos Zaroliagis[1,2]

[1] Dept of Computer Eng. & Informatics, Univ. of Patras, 26504 Patras, Greece
{gianakok,zaro}@ceid.upatras.gr
[2] Computer Technology Institute and Press "Diophantus", 26504 Patras, Greece
gioulycs@gmail.com
[3] Computer Science & Engineering Dept., University of Ioannina, 45110 Ioannina, Greece
kontog@cse.uoi.gr

Abstract. We present a cloud-based time-dependent routing service, which is a core component in providing efficient personalized renewable mobility services in smart cities. We describe the architecture of the time-dependent routing engine, consisting of a core routing module along with the so-called urban-traffic knowledge base, which creates, maintains and stores historic traffic data, as well as live traffic updates such as road blockages or unforeseen congestion. We also provide the crucial algorithmic details for providing the sought efficient time-dependent routing service. Our cloud-based time-dependent routing service exhibits an excellent practical behavior on a diversity (w.r.t. to scale *and* type) of real-world road networks.

1 Introduction

The development of efficient route planning services for traveling in smart cities is a highly sought-after commodity nowadays. Such services are delivered to the smartphones of travelers, who pose route planning queries and receive answers in these devices.

Nevertheless, real-world road networks are typically of very large scale, and demonstrate a time-varying behavior. For instance, the traversal-times of the road segments in the network depend strongly on the actual times of the traversal. This in turn makes it hard, if not impossible due to the practically prohibitive storage and computing requirements, for advanced time-dependent (TD) route planning services to run on isolated devices that have limited computational capabilities, such as our portable navigation devices (PNDs) and smartphones. Additionally, even for a typical route planning server, classical route planning algorithms, such as Dijkstra's algorithm, are not an option. Such a server would have to respond to several dozens, or even hundreds of queries per minute, and Dijkstra's algorithm would require a few seconds per query for large-scale, time-dependent instances of road networks, thus making such an approach highly

impractical. For this reason, extremely efficient heuristic approaches (*speedup* techniques, see e.g., [2] and references therein) and approximation algorithms with provable guarantees (*distance oracles*, see e.g., [1,9,10,11,12,13,14,15] and references therein) have been designed, analysed and experimentally tested during the last years, also for time-dependent instances [5,6,7].

Another axis of complexity, for providing route plans in time-dependent road networks, is the fact that the historic traffic data and, most importantly, the traffic metadata which are created by the routing service, are typically extremely demanding in terms of computational capabilities. Moreover, rather than being created and stored only once, they also have to be periodically updated (say, on a weekly basis), according to the aggregated traffic information of actual speed samples provided by the connected travelers to the service. This is an extremely demanding task for a single server. On the other hand, this maintenance task is extremely parallelizable and of varying computational demands, based on the required changes for the updates. Therefore, the *elasticity of a cloud architecture* would allow for the adaptation of the reserved computing resources to the actual demands for preprocessing traffic-related data and metadata.

One more complication is posed by the fact that the characteristics of the real-world road networks, apart from demonstrating a predetermined time-dependent behavior, also have to cope with unpredicted incidents (e.g., temporal blockages of road segments due to construction works, accidents, etc.), which are typically reported by several sources of information, such as the municipality, the police, or even the travelers themselves. This *live-traffic* information has to be interleaved with the historic (time-dependent) traffic information, in order for the time-dependent routing (TDR) service to provide live-traffic aware routes to the travelers.

All these crucial challenges necessitate the adoption of more sophisticated TDR architectures, which are able to both digest very large amounts of historic information, and also continuously interact with live-traffic sources of information in real time. The heart of such a sophisticated TDR service would have to lie on a cloud architecture, which would be able to guarantee data persistence, interoperability with other sources of traffic-related information, real-time elasticity of computing resources, and also transparent accessibility of the TDR service by the travelers. In such an environment, the queries are sent to a routing engine residing at the cloud infrastructure, which in turn sends back the answers taking into account the updated historic and live-traffic information which is at its disposal.

In this paper, we make two contributions: (i) we describe the architecture of a cloud-based TDR engine that consists of a core module, the so-called *urban-traffic knowledge base* (UTKB), whose role is to create and periodically maintain historic traffic data and metadata, and also to digest in real-time live-traffic update data that are spontaneously provided by diverse sources of information; and (ii) we provide the algorithmic details for providing the sought-after efficient TDR service that exploits all this periodically processed traffic-related

information along with the live-traffic updates, in order to respond in real-time to arbitrary route planning queries.

The specific cloud-based architecture of our TDR service constitutes part of a broader cloud-based platform, developed in the frame of [8], whose aim is to provide a live *community of travelers*, equipped with an arsenal of interoperable personalized renewable mobility services, for large-scale urban road networks. Extensive experimentation with real-world data sets (road networks of Berlin and Germany) demonstrated an excellent performance of the core TD algorithmic routing engine [5]. The specific cloud-based TDR service has been also tested in a pilot phase (in the frame of [8]) in the city of Vitoria-Gasteiz, demonstrating very efficient practical behavior.

The rest of the paper is organized as follows. Section 2 presents the formal problem setting along with the necessary definitions and notation. Section 3 presents the architecture of the TDR service that involves the details of the TDR algorithmic engine, the UTKB, as well as the digestion of unforeseen live-traffic (e.g., emergency) reports and traffic prediction alerts. Section 5 presents the results of the application of our TDR service on a real-world environment. We conclude in Section 6.

2 Preliminaries

In this section, we provide the necessary definitions and notation that will be used throughout the paper adopted from [6,7]. For any integer $k \geq 1$, let $[k] = \{1, 2, \ldots, k\}$. We consider the classical modeling of a road network as a directed graph $G = (V, A)$, with $|V| = n$ nodes or vertices, and $|A| = m \in \mathcal{O}(n)$ arcs (as is the typical case of such networks). Nodes represent road crossings and an arc $a = (u, v)$ between two nodes u and v represents a road segment between two road crossings (without any other crossing intervening between them).

Every arc $a \in A$ is accompanied with a continuous, periodic, pwl *arc-traversal time* function defined as follows: $\forall k \in \mathbb{N}, \forall t \in [0, T),\ D[a](kT + t) = d[a](t)$, where $d[a] : [0, T) \to [1, M_a]$ such that $\lim_{t \uparrow T} d[a](t) = d[a](0)$, for some fixed integer M_a denoting the maximum possible travel time ever seen at arc a. Let also $M = \max_{a \in A} M_a$ denote the maximum arc-traversal time ever seen in the entire network. The minimum arc-traversal time value ever seen in the entire network is also normalized to 1. Since every $D[a]$ is periodic, continuous and pwl function, it can be represented succinctly by a number K_a of breakpoints defining $d[a]$. Let $K = \sum_{a \in A} K_a$ denote the number of breakpoints to represent all the arc-traversal time functions in G, $K_{\max} = \max_{a \in A} K_a$, and let K^* be the number of *concavity-spoiling* breakpoints, i.e., the ones in which the arc-delay slopes increase. Clearly, $K^* \leq K$, and $K^* = 0$ for *concave* arc-traversal time functions.

The *arc-arrival-time* function of $a \in A$ is defined as $Arr[a](t) = t + D[a](t)$, $\forall t \in [0, \infty)$. The *path-arrival-time* function of a path $p = \langle a_1, \ldots, a_k \rangle$ in G (represented as a sequence of arcs) is the composition of the arc-arrival-time

functions for the constituent arcs:

$$Arr[p](t) = Arr[a_k](Arr[a_{k-1}](\cdots(Arr[a_1](t))\cdots)) .$$

The *path-travel-time* function is then $D[p](t) = Arr[p](t) - t$. For any pair of vertices $(o, d) \in V \times V$, let $\mathcal{P}_{o,d}$ be the set of *od*-paths in G.

The *earliest-arrival-time* and *shortest-travel-time* functions are defined as follows: $\forall t_o \geq 0$,

$$Arr[o,d](t_o) = \min_{p \in \mathcal{P}_{o,d}} \{Arr[p](t_o)\}$$
$$D[o,d](t_o) = \min_{p \in \mathcal{P}_{o,d}} \{D[p](t_o)\} = Arr[o,d](t_o) - t_o.$$

The set $SP[o,d](t_o) = \{p \in P_{o,d} : Arr[p](t_o) = Arr[o,d](t_o)\}$ is the set of shortest-travel-time paths for the *query* (o, d, t_o).

A $(1+\varepsilon)$-*upper-approximation* $\overline{\Delta}[o,d]$ and a $(1+\varepsilon)$-*lower-approximation* $\underline{\Delta}[o,d]$ of $D[o,d]$, are continuous, pwl, periodic functions, with a (hopefully small) number of breakpoints in $[0, T)$, such that the following inequalities hold: $\forall t_o \geq 0$, $\frac{D[o,d](t_o)}{1+\varepsilon} \leq \underline{\Delta}[o,d](t_o) \leq D[o,d](t_o) \leq \overline{\Delta}[o,d](t_o) \leq (1+\varepsilon) \cdot D[o,d](t_o)$.

3 Architecture of the TDR Service

The *time-dependent routing* (TDR) service aims at supporting the TD route-planning functionality for vehicles (be it conventional cars, or electric vehicles), where the typical optimization criterion is the minimization of the total travel-time. In particular, the TDR service is responsible for executing several types of queries made by users. Based on the user requirements or/and preferences, it computes one or several routes which satisfy at least one optimization criterion (such as distance, travel time, fuel consumption, eco-friendliness) and use at least one transportation mode (bus, train, EV, bicycle), at specific departure times or time windows.

An overview of the overall architecture of the TDR service, along with its interaction with other services of the broader cloud system of [8], is depicted in Fig. 1. The TDR service consists of data and metadata structures, mechanisms for creating and maintaining these structures, as well as algorithms that answer user route planning requests in real-time by exploiting the stored data and metadata.

In particular (cf. Fig. 1), the *raw-traffic data* (RTD) is a collection of sequences of breakpoints, one per arc in the network. Each breakpoint is a pair of departure-time (from the tail) and traversal-time (up to the head). The *urban-traffic knowledge base* (UTKB) consists of mechanisms for creating and maintaining the RTD structure. Apart from that, in order for the travelers' PNDs and smartphones to be able to provide elementary route plans even when there is no connection to the cloud, the UTKB aggregates the RTD structure into (static) traffic snapshots (SNAP), which are to be stored in the travelers' devices, as a means of contingency plan in case of loss of connectivity. In support

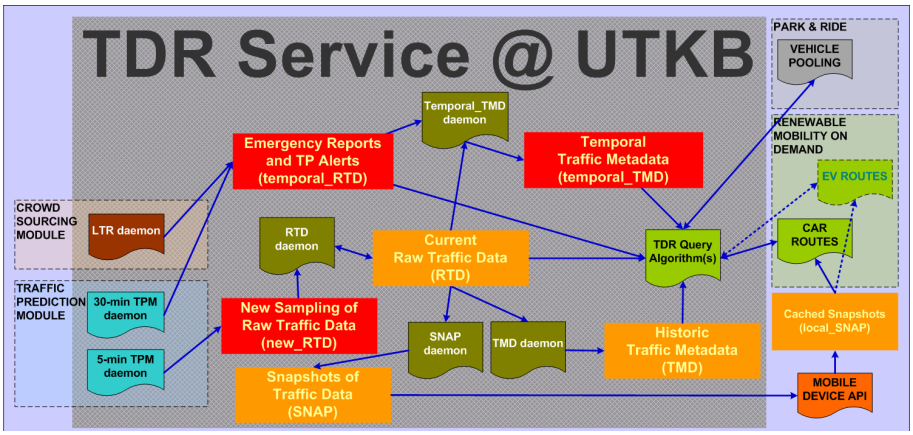

Fig. 1: Overall architecture of the Time-Dependent Routing (TDR) service.

of the sophisticated route planning algorithms (cf. Section 4.2), relevant traffic metadata (TMD) is created and periodically updated. This structure concerns the succinct representation of the (approximate) min-travel-time functions from selected vertices (called *landmarks*) to reachable destinations from them. The UTKB also provides the appropriate procedures to digest live-traffic reporting, i.e., for creating and maintaining temporal RTD and temporal SNAP structures. These are succinct representations (i.e., sequences of breakpoints) of approximate min-travel-time functions from landmarks, which are only effective for a given time-window (depending on the duration of the reported incident). Last but not least, the TDR service also hosts the core TDR query algorithms (its most vital part) that exploit all the aforementioned traffic-related data and metadata.

As mentioned above, all these TDR functionalities, be it data-maintenance procedures or route-planning query algorithms, constitute the urban traffic knowledge base (UTKB) which resides on one or more computing resources of the cloud, depending on the actual computational demands of these functionalities. In the rest of this section, we provide the architectural details of UTKB.

3.1 Urban Traffic Knowledge Base

All the time-dependent urban traffic information and live-traffic monitoring information is organized and maintained in a periodically updated and dynamically-evolving urban-traffic knowledge base (UTKB), so as to support responses to route-planning queries in real time. The UTKB is responsible for the creation and maintenance of traffic-related metadata, to be exploited by route planning and mobility-on-demand services supported by the overall cloud platform. Its main purpose is to handle the periodically changing urban traffic information, by dynamically updating the preprocessed traffic data kept in the system, when needed.

The UTKB receives input mainly by two system modules, the *Crowd-sourcing Module* (CM) and the *Traffic Prediction Module* (TPM), which are used as black-box services and are responsible for creating and assessing the traffic-related information received either by the road network itself, or by the travelers, before sending appropriate update signals to the UTKB.

More precisely, the incoming data may concern *periodic traffic reports* (e.g., traversal-time samples of road segments, every 5 minutes), *emergency reports* (e.g., spontaneous reports of accidents, predictions of unforeseen evolution of congestion in particular road segments, etc.), information on changes of external parameters (e.g., weather conditions), updates on public-transport's mobility plans, or updates on publicly used EVs' availability information. Each of these reports actually demands for an appropriate update of the involved traffic-related information. Both the range of affection of a newly reported incident (within the network) and the temporal traffic-related metadata in response to this particular report, has to be determined by the UTKB itself. As a result, the historic traffic data and metadata kept within the UTKB is interleaved with the temporal traffic metadata created per reported incident, but only for those routes which are indeed affected by it, so that live-traffic aware responses to arbitrary route-planning queries are provided by the query algorithms in real time.

As previously mentioned, the UTKB also creates and periodically maintains snapshots of the current traffic status (that is, average arc-traversal time *values* rather than time-dependent arc-traversal time *functions*, e.g., for rush-hours or free-flow route plans, depending on the traveler's departure time), to be at the disposal of the travelers for downloading them to their personal devices, so as to assure a minimum level of the routing service even without connection to the cloud system.

The main purpose of the UTKB is to gather and digest all the (spontaneous, or periodic) observations of the live-traffic situation, and dynamically update its contents. The input data and the corresponding update actions applied to them can be divided into seven basic categories: (i) user generated periodic speed reports, mainly concerning private cars and EVs; (ii) information on public-transport data (that includes static information, such as timetables, but also planned events, e.g. a subway station must be closed due to maintenance, as well as live traffic data, where in particular, delays are important); (iii) energy-consumption information (e.g., current state-of-charge) concerning EVs; (iv) spontaneously provided emergency reports, which include unpredictable traffic disruptions (i.e., currently unavailable road segments, changes on weather conditions, etc.), reported either by a (totally reliable) public authority, or by (possibly unreliable) travelers whose credibility is based on a wisdom-of-the-crowd approach (the more travelers reporting an incident, the more likely it is that it is actually true); (v) short-term traffic prediction alerts, as reported by the TPM, acting as periodic travel-time samples (say, every 5 minutes) of the entire network; (vi) long-term traffic prediction alerts, as reported by the TPM (say, every 30 minutes), in order to automatically detect unforeseen evolution of the traffic pattern and provide appropriate signals (analogous to the

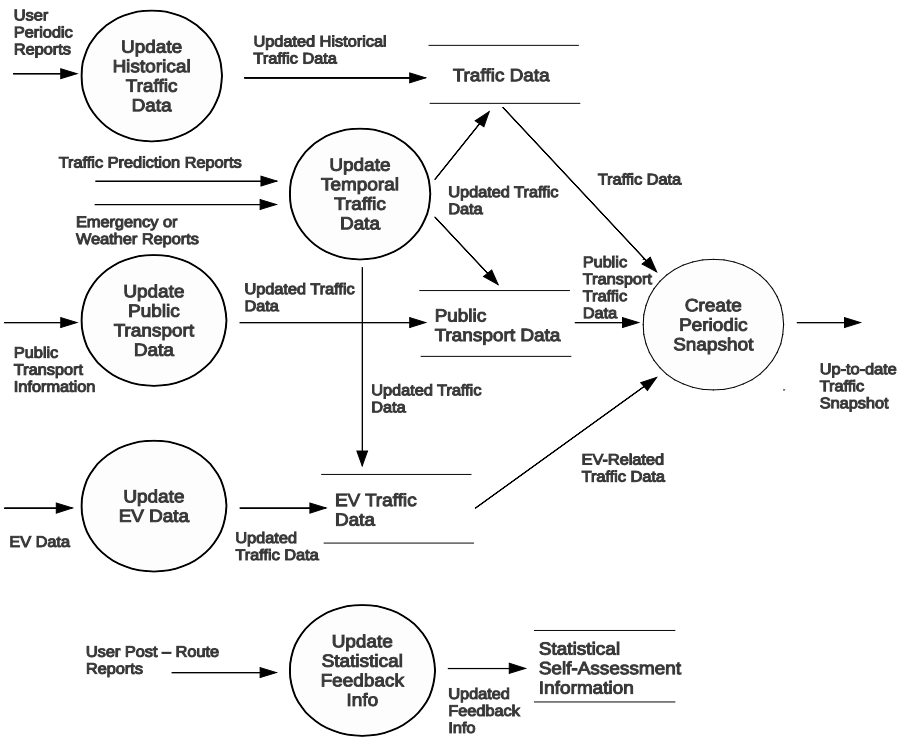

Fig. 2: Detailed DFD for the UTKB Architecture.

emergency reports) whenever the predicted behavior of a road segment deviates significantly from the behavior described in the historic raw data; and (vii) users' post-route spontaneous reports, containing route-related information, or a kind of like/dislike feedback on the routes recommended to them, as a means of self-assessment of the TDR service.

In each case, an appropriate update on the historical or the temporal traffic data and metadata kept in the system must be performed by the UTKB. A corresponding bubble, describing each update process, is added in a detailed data flow diagram (DFD) of the UTKB, shown in Fig. 2. All the aforementioned types of information are carefully collected and stored in the UTKB, therefore a data storage is displayed for each type of data. Additionally, a unique process should take over the responsibility of periodically creating a snapshot of the current traffic status, which will be available for downloading to the users' portable devices, upon their own request.

A second-level refinement of UTKB's architecture is accomplished by mapping individual bubbles or groups of bubbles on the same side of a boundary of the DFD into appropriate modules within the UTKB's architecture. Four main sub-modules of the UTKB module are considered: (i) Historical Traffic Data Up-

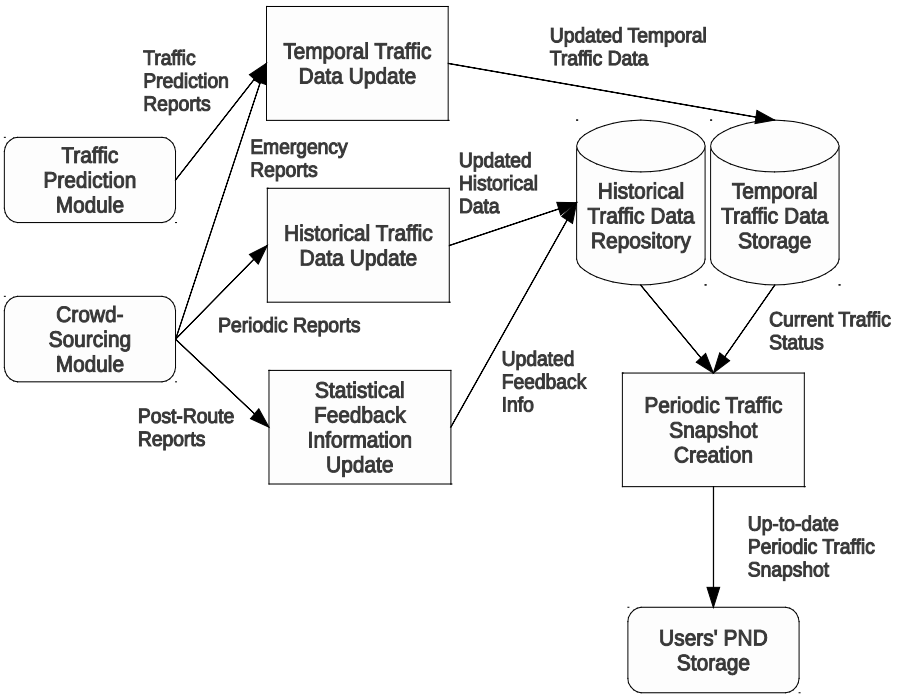

Fig. 3: The high-level architecture diagram for UTKB Module.

date; (ii) Temporal Traffic Data Update; (iii) Statistical Feedback Information Update; and (iv) Periodic Traffic Snapshot Creation. The data that the UTKB module keeps and handles, either historical or temporal, may concern all types of transport modes, be it private cars, EVs or public-transport modes. Based on the resulting sub-modules, the corresponding hierarchical schema turns out to be too simple. The result is depicted in Fig. 3, which shows the high-level architectural diagram of the UTKB.

4 Functionality of the Cloud-based TDR Service

In this section, we describe the functionality of the TDR service for computing routes in road networks, concerning mainly private cars.

4.1 Creation and Maintenance of Traffic Metadata

We start with the algorithmic technique used for the actual creation of the traffic metadata (TMD) concerning private cars, as well as the methodology adopted for the efficient maintenance of all the traffic-related information kept

in the UTKB, so that it can be continuously available and exploited by the TDR service residing at the UTKB server. The theoretical analysis (correctness and complexity bounds) of the algorithmic technique is provided in [6].

The TDR service is able to respond in real time to arbitrary shortest-path queries, by computing an optimal origin-destination path providing the earliest-arrival time at a destination, when departing at a given time from the origin. In order for the route-plans given by TDR to be fast and accurate, the service exploits the appropriately selected shortest-path information, precomputed off-line. In particular, a carefully selected subset of vertices (*landmarks*) is equipped with succinct representations of shortest-travel time *functions* to all other vertices in the network. Apart from creating these traffic metadata, the UTKB has also to update them periodically, according to the periodical adaptations of the historic traffic data kept in it. Both the creation and the periodic updating of all this landmark-related traffic metadata, are extremely time-consuming tasks whose exact computational needs are indeed unclear.

The role of the cloud, which allocates the appropriate amount of processing power depending on the work to be done, is actually crucial at this preprocessing phase which has to be completed within a few hours and is repeated frequently (e.g., once per week). Since computing and storing the exact shortest-travel time functions turns out to be hard (in particular, super-polynomial) [3], we compute $(1+\varepsilon)$-approximations of these functions, called *travel-time summaries*, from the selected landmarks towards all other vertices. The main challenges of this major task are: (i) to ensure that the preprocessing time and space complexity is actually manageable (e.g., subquadratic in the size of the network); (ii) to allow for route-planning query algorithms which provide fast responses, both in theory and in practice; and (iii) to obtain provably good approximation guarantees.

Approximating travel-time summaries. Our main building block for the preprocessing is an approximation technique for the computation of all landmark-to-vertex approximate travel-time summaries. We briefly describe here a novel method, the *trapezoidal technique* (TRAP), which is a one-to-many method that concurrently computes travel-time summaries from a given landmark to many (or even all) destinations which are reachable from it.

For a given set of landmark-nodes L (whose choice will be determined later), our goal is to construct all the $(1+\varepsilon)$-upper-approximation shortest travel-time functions (travel-time summaries) from each landmark towards all reachable destinations, for a time period of a day, i.e., in the interval $[0, T = 86400\text{sec})$.

Instead of computing the exact minimum-travel-time (which are continuous, pwl and periodic) functions $D[\ell, v]$ from each $\ell \in L$ towards each reachable $v \in V$, we seek for their $(1+\varepsilon)$-upper-approximations $\overline{\Delta}[\ell, v]$ (the travel-time summaries). Recall from Section 2 that $\overline{\Delta}[\ell, v]$ and $\underline{\Delta}[\ell, v]$ denote $(1+\varepsilon)$ upper- and lower-approximations of $D[\ell, v]$ which are also continuous, pwl, periodic functions, hopefully with a small (in particular, independent of the size of the network) number of breakpoints in [0,T), such that the following inequalities

hold: $\forall t_\ell \geq 0$,

$$\frac{D[\ell,v](t_\ell)}{(1+\varepsilon)} \leq \underline{\Delta}[\ell,v]](t_\ell) \leq D[\ell,v](t_\ell) \leq \overline{\Delta}[\ell,v](t_\ell) \leq (1+\varepsilon)D[\ell,v](t_\ell)$$

The *trapezoidal method* (TRAP) is a one-to-all approximation algorithm for computing concurrently all functions $\overline{\Delta}[\ell,v]$ for a given landmark ℓ and all reachable destinations $v \in V$. The theoretical analysis of TRAP can be found in [6]. TRAP splits the entire period $[0,T)$ in small (length-τ) subintervals, and within each of them, say $[t_s, t_f = t_s+\tau) \in [0,T)$, it provides the appropriate projection of $\overline{\Delta}[\ell,v]$, by essentially exploiting the fact that $\tau > 0$ is indeed small, along with the following assumption on the travel-time slopes of all minimum-travel-time functions in the network:

Assumption 1 (Bounded Travel-Time Slopes) *All min-travel-time slopes are bounded in a given interval* $[-\Lambda_{\min}, \Lambda_{\max}]$*, for* $\Lambda_{\min} \in [0,1)$ *and* $\Lambda_{\max} \geq 0$*.*

The validity of this assumption has been experimentally verified in real-world data sets that we have at our disposal [5].

Within each subinterval $t_s, t_f = t_s+\tau)$, TRAP provides a crude approximation of the unknown function $D[\ell,v]$, concerning the minimum slope $-\Lambda_{\min}$ and maximum slope $\Lambda_{\max}$ of the actual shortest-travel-time functions in the instance. In particular, for every subinterval $[t_s,t_f)$, TRAP works as follows. For the two boundary departure times t_s and t_f, we sample concurrently (by making two calls to the time-dependent variant of Dijkstra's algorithm) the travel-time values for each destination $v \in V$. We then consider the semi-lines with slope $\Lambda_{\max}$ from t_s and $-\Lambda_{\min}$ from t_f. The upper-approximating function $\overline{\Delta}[\ell,v]$ that we consider within $[t_s,t_f)$ is the lower-envelop of these two semi-lines. Analogously, the lower-approximating function $\underline{\Delta}[\ell,v]$ is the upper-envelop of the semi-lines that pass through t_s with slope $-\Lambda_{\min}$, and from t_f with slope $\Lambda_{\max}$. In particular, TRAP considers the following two (upper- and lower-) approximating functions of $D[\ell,v]$ for every possible departure time $t \in [t_s,t_f)$:

$$\overline{\Delta}[\ell,v](t) = \min\left\{ D[\ell,v](t_s) - \Lambda_{\max}t_s + \Lambda_{\max}t, D[\ell,v](t_f) + \Lambda_{\min}t_f - \Lambda_{\min}t \right\}$$

$$\underline{\Delta}[\ell,v](t) = \max\left\{ D[\ell,v](t_f) - \Lambda_{\max}t_f + \Lambda_{\max}t, D[\ell,v](t_s) + \Lambda_{\min}t_s - \Lambda_{\min}t \right\}$$

Considering $\overline{\Delta}[\ell,v]$ as the required travel-time summary for departure-times in $[t_s,t_f)$, the algorithm has to decide whether this is actually a $(1+\varepsilon)$-upper-approximation of $D[\ell,v]$. For this reason, we must compute the maximum additive error $MAE(t_s,t_f)$, which is the maximum delay-axis distance of the two functions $\overline{\Delta}[\ell,v]$ and $\underline{\Delta}[\ell,v]$ within $[t_s,t_f)$. This is done as follows. Let $(\underline{t}_m, \underline{D}_m)$ be the intersection point of the two legs involved in the definition of $\underline{\Delta}[\ell,v]$. Similarly, let $(\overline{t}_m, \overline{D}_m)$ be the intersection point of the two legs involved in the definition of $\overline{\Delta}[\ell,v]$. The (worst-case) *maximum additive error* $MAE(t_s,t_f)$ guaranteed for $\overline{\Delta}[\ell,v]$ within $[t_s,t_f)$ is:

$$\begin{aligned} MAE(t_s, t_f) &:= \max_{t \in [t_s, t_f)} \left\{ \overline{\Delta}[\ell, v](t) - \underline{\Delta}[\ell, v](t) \right\} \\ &= \overline{\Delta}[\ell, v](\underline{t}_m) - \underline{\Delta}[\ell, v](\underline{t}_m) \\ &= \overline{\Delta}[\ell, v](\overline{t}_m) - \underline{\Delta}[\ell, v](\overline{t}_m) \end{aligned}$$

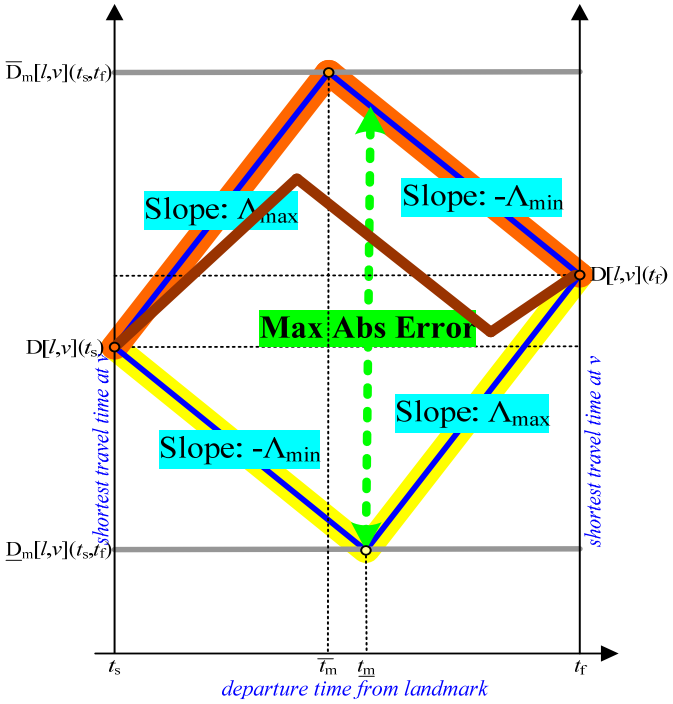

Fig. 4: The upper-approximating function $\overline{\Delta}[\ell, v]$ (the orange, upper pwl line), and lower-approximating function $\underline{\Delta}[\ell, v]$ (the yellow, lower pwl line), of the unknown distance function $D[\ell, v]$ within $[t_s, t_f)$.

Fig. 4 provides a visualization of all the above mentioned quantities, as well as the upper- and lower-approximating functions returned by TRAP within $[t_s, t_f)$.

For each subinterval $[t_s, t_f)$, the algorithm checks whether the constructed upper-approximating function $\overline{\Delta}[\ell, v]$, valid for every possible departure time $t \in [t_s, t_f)$ from the origin ℓ, is actually a $(1 + \varepsilon)$-upper-approximation of the exact shortest-path function in the same interval. If this is the case, $\overline{\Delta}[\ell, v][t_s, t_f)$ is accepted. Otherwise, the length of the sampling interval τ needs to be even smaller. TRAP handles all possible sampling intervals as follows.

Rather than splitting the entire period $[0, T)$ in a flat manner, i.e., into equal-size intervals, we start with a coarse partitioning based on a large length τ and then in each interval and for each destination vertex we check for the provided approximation guarantee by TRAP. All the vertices which are already satisfied by this guarantee with respect to the current interval, become inactive for this and all its subsequent subintervals. If there is at least one active destination vertex, for which the function $\overline{\Delta}[\ell, v]$ constructed in the current interval violates the maximum absolute error, we proceed by splitting in the middle the current subinterval, and repeat the check within the new subintervals created. The algorithm terminates when all reachable destinations become inactive for every subinterval of $[0, T)$, which means that every one of them has a $(1 + \varepsilon)$-upper-approximation function for the entire period.

We keep every constructed $\overline{\Delta}[\ell, v]$ function as a sequence of sampled breakpoints, which are of the form (*departure-time*, *travel-time*, *approximate-path-predecessor*). The predecessor of v results from the sampling that the algorithm performs at each subinterval $[t_s, t_f)$. To reduce the required space in memory, we do not store the node-id of each predecessor. Instead, we only need to keep a small integer indicating the index of the corresponding incoming arc of v in the adjacency list. We have further developed a host of heuristic algorithmic improvements to boost performance in practice [5]; for completeness, we describe them in Appendix A.

Maintenance of Traffic Metadata. The traffic metadata produced by TRAP are efficiently maintained in the UTKB server, according to the following methodology. In order to access the data blocks of each landmark in $O(1)$ time, we use a mapping. For retrieving efficiently each approximate travel-time function from a landmark to any destination vertex, we need to store an index. In particular, for each landmark ℓ we maintain a vector of pointers, the size of which equals to the number of destinations. The order of the pointers is in ascending order of node id and each one of them corresponds to a destination v. Thus, the address of the $\overline{\Delta}[\ell, v](t)$ data is provided in $O(1)$ time, while the required space for this indexing is $O(n \cdot |L|)$.

To keep the preprocessing space small, we store the information about both predecessors using the same unit of memory. In the case that the approximate function from a landmark ℓ to a destination v is constant, we expect the approximate predecessor of v to be the vertex ℓ and the predecessor corresponding to the piecewise composition (not performed in this case) to be the destination vertex v itself.

We describe next how the appropriate traffic metadata related to every day of the week are uploaded in the UTKB server, so that they are available to the TDR service. A continuously running TDR daemon is responsible for this task. In particular, at the beginning of each day the corresponding traffic-data is uploaded, while the data related to the previous day is automatically removed.

Finally, the TDR daemon also undertakes the creation of new updated TMD for any day in an off-line fashion (e.g., at the end of each day). In particular, the

TPM module (used as a black-box daemon service, also residing at the cloud) provides the TDR daemon with periodic (say, per 5 minutes) predictions of travel-time value estimations, for all the road segments in the network. All these samples constitute, by the end of the day, a fresh image of the historic data for the entire day, which is then aggregated by the TDR daemon in the current historic data (with an appropriately small weight, so as to avoid over-fitting or oscillation effects), so that the historic raw traffic data residing at the UTKB converge to the actual travel-time functions of all the road segments, per day.

4.2 TDR Query Algorithms

The TDR service is developed to provide significantly fast and accurate min-cost route plan responses to arbitrary shortest-path queries, exploiting (i) a carefully selected landmark-set of vertices; (ii) the historical and temporal traffic-related information kept in the UTKB server; and (iii) an efficient approximate query algorithm, designed to provide the required routes. In this section, we describe how the query algorithms supported by the routing service work, as well as how the exploitation of the historical traffic-data and those provided by the Traffic Prediction Module and the Crowd-sourcing Module is achieved.

The daemon residing at the UTKB server continuously runs and accepts incoming origin / destination / departure-time shortest-path queries (o, d, t_o). For each one of them, the query algorithm provided by the routing service is called and returns either the exact minimum-travel-time value, along with the corresponding o-d path, or an approximate travel-time value via an appropriate landmark-node ℓ and the corresponding approximate o-ℓ-d path.

Three query algorithms have been implemented and extensively tested. We describe them below. Their theoretical analysis (correctness and complexity bounds) is provided in [6,7].

The first one, which is called FCA, grows a Time-Dependent-Dijkstra (TDD) ball from (o, t_o) until either d or the closest landmark l_o is settled. In the former case it returns the minimum travel-time value and the corresponding shortest path. In the latter case, the $(1 + \varepsilon + \psi)$-approximate travel-time value of an o-d path passing by l_o is returned, where ψ is a constant that depends on characteristics related with travel-times, but is independent of the size of the network.

The second algorithm, called FCA$^+(N)$, is a variant of FCA which keeps growing a TDD ball from (o, t_o) until either d or a given number N of landmarks is settled. FCA$^+$ then returns the exact travel-time value, or the smallest via-landmark approximate travel-time value, among all these settled landmarks. Theoretically, the approximation guarantee of FCA$^+$ is the same as that of FCA, but its practical performance is actually impressive [5].

The third algorithm, called RQA, improves the approximation guarantee provided by FCA, by exploiting carefully a number r (called the *recursion budget*) of recursive accesses to the preprocessed information, each of which produces (via calls to FCA) additional candidate o-d paths. RQA works as follows. As long as the destination vertex within the explored area around the origin has not yet been discovered, and there is still some remaining recursion budget, it

"guesses" (by exhaustively searching for it) the next vertex w_k of the boundary set of touched vertices (i.e., still in the priority queue) along the unknown shortest o-d path. Then, it grows an outgoing TDD ball from every new center $(w_k, t_k = t_o + D[o, w_k](t_o))$, until it reaches the closest landmark ℓ_k to it, at travel-time $R_k = D[w_k, \ell_k](t_k)$. Every new landmark offers an alternative o-d path by a new application of FCA for every boundary center w_k. RQA finally responds with a $(1 + \sigma)$-approximate travel-time to the query, for any constant $\sigma > \varepsilon$.

The response-times as well as the approximation guarantees provided by all three query algorithms have a strong dependence on the selected landmark-set [5]. Our experimental study [5] has shown that FCA has the fastest performance (as expected), and provides quite small approximation guarantees. Both RQA and FCA$^+$ are fast, i.e., they run in time less than 1 msec, but also significantly accurate, since they produce solutions with relative errors less than 1% in the average case. FCA$^+$ is in some cases better than RQA regarding accuracy, while it is almost as fast as RQA. In fact, we can control the trade-off between time and accuracy, by selecting a smaller or larger number of landmarks to be discovered by the algorithm. For those reasons, FCA$^+$ is the default algorithm running in the TDR service.

Live-traffic awareness and route computation. We describe in detail the way in which the query algorithm manages to exploit the historical traffic-data as well as the temporal data corresponding to live-traffic updates, and finally provides the resulting route as output, after the application of a path reconstruction method, for retrieving the unknown approximate paths.

For any incoming shortest-path query, the algorithm considers the flags associated with all arcs and landmarks in the network, to indicate whether there exists active temporal data for them or not. Any temporal traffic-updates have to be adopted. As described in Section 4.3, if there are any road segments (and the relevant raw-traffic data), or landmark-nodes (and the associated traffic metadata maintained by the UTKB) affected by an emergency report or traffic prediction alert, a flag corresponding to those arcs and landmarks in the network is raised, to indicate that for a specific time-window the temporal RTD and temporal TMD structures have to be considered, rather the original historic information kept in the RTD and TMD structures, respectively. The query algorithm absorbs the live-traffic changes, by taking into account these flags on the affected arcs and landmarks.

More precisely, we consider the two basic phases of any query algorithm, i.e., the first step, which is a TD Dijkstra-based search, and the second one, which retrieves the approximate distance from a settled landmark-vertex to the destination, by searching into the appropriate preprocessed distance function. For the first step, we perform the following modification on the relaxation of arcs. Each time that we need to compute the travel-time needed to traverse an arc, given a departure-time from its tail, the algorithm checks the flag corresponding to the arc, so as to search for its travel-time function either in the current or

the temporal RTD. We need to take the temporal raw traffic-data into account, in the case that a specific arc was recently affected by a live-traffi update, for a particular time-window around the departure time from its tail. The update-daemon running on the UTKB server is responsible to modify the temporal RTD for all affected arcs and raise the bit flag on them so long as an update has been adopted and is still active. Both the historic and the temporal travel-time functions are stored per arc.

The second phase of the algorithm adopts a similar modification for landmarks. If the destination was not discovered by the first step, the algorithm collects one or more landmarks, each providing a distinct approximate solution towards the destination. If a landmark-node has been affected by a live-traffic update, we have to store the time window of corresponding departure times from it for which the update will still be active, as well as a pointer to the address of the corresponding temporal TMD for this landmark. For each landmark, the algorithm checks whether its update-flag bit is 0 or not, so as to decide if the specific landmark has active temporal traffic-data for a particular time window, and therefore there exists an updated approximate distance function from it, kept at the appropriate memory block. The temporal TMD are considered for a landmark ℓ when its update-flag bit is still 1 upon arrival at ℓ, meaning that there is a pointer to a memory address which is not NULL, and the departure time from ℓ is within an affected time window.

By adopting these simple modifications, we are able to exploit the real-time traffic conditions of the network and provide fast and accurate responses to route-planning requests.

Finally, we describe the *Path Reconstruction* method followed for the generation of the computed o-d path, as a sequence of arcs. In the case that the shortest path returned by the query algorithm is exact, i.e., the destination was discovered during the TDD-search (which is actually quite possible to occur), the path is constructed by simply following the predecessors from the destination to the origin, which the TDD ball provided. However, if the algorithm decides that the destination is to be reached via an appropriate landmark ℓ, we need a method to retrieve the unknown sequence of predecessors from the destination up to the landmark, corresponding to the approximate ℓ-d path. For this purpose, we exploit the information kept in all TMD. For every possible destination d and for any departure time from ℓ, the travel-time function contains the immediate predecessor of d, valid for a specific time interval, where TRAP performed its sampling. Based on this information, along with some heuristic ideas, the path reconstruction works as follows.

Let v denote every node that belongs on the path we want to construct. We start with $v = d$. We then obtain the immediate (approximate) predecessor, $approxPred(v)$, given by the approximate travel-time function $D[\ell, d](t)$, when departing from ℓ at time $t_\ell = Arr[o, \ell](t_o)$, which denotes the arrival time set by the Dijkstra-ball. We then mark node v as visited, we set $v = approxPred(v)$ and repeat. The procedure terminates when we reach the landmark ℓ, i.e., $v = \ell$, or at least some already settled vertex by the first time-dependent Dijkstra (TDD) ball

grown from (o, t_o). The retrieval of all predecessors is done by searching either the historical or the temporal TMD, depending on the flag that the algorithm previously considered for ℓ.

In practice, we observed the following phenomena which we tackled accordingly. Firstly, as we reversely approach the landmark ℓ, the sequence of nodes v at some point enters the area explored by the query algorithm. We decided to collect all those explored nodes v and compute the total travel-time $D[v, d](t_v)$, exploiting the reverse arc-traversal time functions on all arcs connecting all nodes v up to that point. When the main loop of our method terminates, we check which explored node on the path we constructed (including the landmark ℓ) gives the minimum $D[o, v](t_o)+D[v, d](t_v)$. This means that there can be some cases where we construct the approximate path via an appropriate explored node v of the TDD ball and not the proposed landmark ℓ, which mainly acts as an "attractor" during the path reconstruction phase, rather than an actual intermediate node of a candidate o-d path.

Next, we observed that a predecessor given by the travel-time summaries stored in UTKB can in fact be already visited, which means that a cycle is possibly created. This can be expected since we search different approximate functions $\overline{\Delta}[\ell, v](t)$ for each vertex v, departing from ℓ. The TRAP method samples the exact travel-time-function in different subintervals for each destination. We choose to face this case as follows. The path reconstruction method returns to its initial step, where $v = d$. Instead of departing from landmark ℓ at the exact departure time t_ℓ, we seek for the closest departure time to t_ℓ contained as a breakpoint in *all* approximate distance functions of the predecessors involved to an approximate path up to ℓ. This safely means that this departure time is a sampling time for all those destinations and thus, they all belong to the very same shortest-path tree, created by the TRAP method during the preprocessing. To avoid the cycle (which in practice is a rare case), we consider the sequence of vertices created, considering the above departure-time from ℓ, which is usually very close to the actual.

After the sequence of predecessors has been constructed, the method simply walks on the arcs connecting them and the ℓ-v path-travel-time value is provided by the (actual) path-travel-time function, when departing from ℓ at its actual arrival-time, set by the algorithm. The path-travel-time function can be provided by historic or temporal raw traffic-data, depending on the flags kept on each arc. The resulting value is at most the approximate one. In practice, we obtain a much better travel-time value. The last step is to connect the constructed ℓ-v path with the exact o-ℓ path, given by the query algorithm, leading to a total o-d route-response, which is usually very close to the exact one.

4.3 Adaptation to Emergency Reports and Traffic Prediction Alerts

The TDR service is responsible for computing shortest paths with respect to the current status of the network. In such a service that responds to several queries in real-time, various disruptions may occur "on the fly" (e.g., unforeseen congestion, or even blockage of a road segment). In such a case, the new traffic

conditions have to be absorbed in real time and they have to be taken into account by the TDR.

The update of the arc-traversal time functions and the landmark travel-time summaries which are used by the query algorithms is assigned to an online update-daemon worker within TDR. The daemon performs three tasks, whose main functionalities are detailed in the following.

Periodic inspection for Emergency Alerts (EMAs). The arc-traversal time changes in the network are supplied by reports. These are produced by the TPM and they are stored in a file called `alerts.csv`, within UTKB. The daemon periodically reads the file (every 15 minutes), in intervals in which the TPM-daemon does not write. For preventing the infrequent case of reading the file when is still being written by the TPM-daemon, its last modification time in the file system is probed. If it is different before and after the reading phase, then the reading phase is repeated after 1 min. If `alerts.csv` is not empty, then the update-daemon loads in TDR the affected arcs which have to be updated.

Network data update. The TDR must work continuously in order to answer to any user or service query. On this requirement, the update steps have to be performed independently without interrupting the TDR. Under normal conditions, the arcs which need to be updated are few. Therefore the chance for an update not be absorbed in the shortest path computation is small. Also, in the worst case, since the update can be completed in a few milliseconds, running shortly a new query will eventually output the updated shortest paths.

During the reading phase of the disrupted arcs from `alerts.csv`, the daemon inserts them in a queue. Then, it runs the update process for each such arc $a = (u, v)$. Let $T = (t_s, t_e]$ be the affection interval associated with an affected (closest) landmark ℓ and related to the disruption occurred at arc a. During that interval, the temporal data from ℓ should be taken into account. Let the new traversal-time value along arc a be Δ at the time interval $T = (t_s, t_e]$, and the original arc-traversal time function be

$$travelTime_a(t) = \begin{cases} t \cdot \text{slope}_1 + \text{offset}_1, & [t_0, t_1) \\ t \cdot \text{slope}_2 + \text{offset}_2, & [t_1, t_2) \\ \quad \ldots & \ldots \\ t \cdot \text{slope}_k + \text{offset}_k, & [t_{k-1}, t_k) \\ \quad \ldots & \ldots \end{cases}$$

The update steps are as follows.

STEP 1: Initially, the daemon inserts the affection expiration time t_e of the new travel time on a in a priority queue PQ. This is because the emergency alerts may concern short-term changes. Consequently, after the expiration of the time-window of affection, the original travel time function of a will be restored.

STEP 2: The daemon generates the updated travel time function of arc a. Initially, it detects the affected legs of $travelTime(t)$ which have to be updated,

within the time interval $[t_s, t_e]$. For example, if the new travel time Δ is occurred in $[t_{k-1}, t_k)$, then the candidate legs to be modified are $(\text{slope}_i, \text{offset}_i)$, $i = k-1, k, k+1$. In such a case, the linear interpolation is applied on the new travel time values throughout the interval $[t_s, t_e]$. The updated travel time function is stored in a different memory address.

In order TDR to be informed about the new travel time function of a, an update-flag bit is associated to a and it is set to 1 only if the creation of the function is finished. Consequently, during a shortest path computation the update-flag bit of a indicates that its travel time function has changed. In addition, if a belongs to a shortest path, then a pointer to the address of the updated travel time function is accessed by the routing service.

STEP 3: The daemon re-computes the travel-time summaries for a subset of landmarks L in the vicinity of the disrupted arc a. In particular, it runs a backward TD-Dijkstra from u (tail-node of a) under the free-flow metric, with travel time radius of $t_e - t_{cur}$, where t_{cur} is the current time (based on the corresponding network's UTC). The travel-time radius is used to trace only the nearest landmarks that may actually be affected by the disruption, leaving unaffected all the "faraway" landmarks. This means that the update has to be performed only for the involved drivers who are close to the area of disruption. For each affected landmark ℓ, we consider a disruption-times window $T_\ell = [d_s, d_e]$, containing the latest departure times from ℓ for arriving at the tail u at any time in the interval $[t_s, t_e)$ in which the disruption occurs. The T_ℓ windows, for all landmarks $\ell \in L$, are computed by running two backward TD-Dijkstra queries from u under the time-dependent-flow metric, the first with arrival-time equal to t_s and the second with arrival-time equal to t_e. We then compute the temporal travel-time summaries for each affected landmark ℓ at its disruption-times window T_ℓ. The produced travel-time summaries are stored in a different memory address.

In order TDR to be informed about the new travel-time summaries of landmark ℓ, an update-flag bit is associated to ℓ and it is set to 1 only if the creation of the travel time summaries is finished. Consequently, during a shortest path computation, if ℓ is required at the specific disruption-time window T_ℓ, the update-flag bit indicates that the travel time summaries of ℓ have changed on T_ℓ. In this case, a pointer with the address of the updated travel-time summaries is accessed by the routing service.

Expiration monitoring. The daemon wakes up from the idle state when the affection expiration time t_e of a disrupted arc is reached, based on the network's UTC. In such a case, it extracts the arc a having the earliest t_e from the priority queue PQ. Then it sets the update-flag bit to 0 on arc a and any corresponding landmark $\ell \in L$ which were updated due to a. Consequently, during a shortest path computation, when it is required, the routing service will use the original travel time function of arc a. Similarly, if a landmark $\ell \in L$ is used on a shortest path computation, then the routing service will use the original travel time

summaries of ℓ. In the end, the daemon removes the outdated temporal travel time functions and summaries.

5 Practical Performance of the TDR service

The TDR cloud-service constitutes part of a broader cloud-based platform, developed in the frame of [8], that delivers personalized services for renewable mobility within cities. In this section, we report on the practical performance of TDR. In particular, (i) we report on an extensive experimental study carried out with real-world data sets from the road networks of Berlin and Germany [5]; (ii) we report on a pilot study of the TDR service carried out in real-world conditions in the city of Vitoria-Gasteiz (in the frame of [8]).

5.1 Experimental Study on Real-world Road Networks

In this section, we succinctly report the outcome of our experiments on the instances of Berlin and Germany (details can be found in [5]). The instance of Berlin consists of $473,253$ nodes and $1,126,468$ arcs, while the instance of Germany consists of $4,692,091$ nodes and $11,183,060$ arcs.

We measured the performance of the basic query algorithms FCA, FCA$^+$ and RQA, with respect to absolute running times and Dijkstra-rank[4] values, respectively, for several types of landmark set selections (for instance, variants of a random selection, and/or partition-based selection of landmarks from the boundary vertices of the partition), and various sizes.

In particular, we considered a uniformly at random selection of landmarks among all vertices, denoted by R, and a random selection of landmarks among the boundary vertices resulted by the graph-partitioning tool KaHIP [4], denoted by K. Two variations of S and K stood clearly above others: the variation SR of R, where each newly chosen random landmark excludes its closest 300 vertices (under the free-flow metric) from being landmarks in the future; and the (similar in nature with SR) variation SK of K.

As for the query algorithms, we used recursion budget 1 for `RQA` and we let `FCA`$^+$ settle the 6 closest landmarks, which is roughly the average number of settled landmarks by `RQA` as well.

For Berlin, our fastest query algorithm, FCA, combined with the SR-landmark set, achieved average response times of 83μsec, with relative error of 0.781%, absolute runtime speedup more than 1146, and Dijkstra-rank speedup more than 1570, compared to TDD. If the relative error is of importance, then one should choose FCA$^+(6)$ along with the SK-landmark set, which achieves 0.616msec query time, and relative error of of 0.226%. In case that space is a main concern, we observed the full scalability in the trade-offs between space and query-responses. For instance, by consuming space 3.2GB, we can achieve query-response time 0.73msec and relative error 2.198% for the Berlin instance.

[4] The Dijkstra-rank of a vertex v is the number of settled vertices by (plain or TD) Dijkstra's algorithm until v is settled.

In Germany, our findings are analogous. Exploiting 6 computational threads, the average preprocessing time is less than 90sec and the average space is up to 25.7Mbytes, per landmark. The best speedup against TDD is achieved by SK-landmarks, and is more than 1531 in Dijkstra-rank, and more than 902 in absolute query-time, with worst-case error at most 1.534%.

The aforementioned results suggest that our TDR service (and in particular its TDR engine) is suitable for practical application. The has indeed happened and described in the next section.

5.2 Pilot Execution in a Smart City

During 2016, the TDR service has been piloted for more than three months in the city of Vitoria-Gasteiz (the most intensive tests were carried our in the period July to October 2016). Two main functionalities of TDR were tested:

1. Its real-time responsiveness to queries for earliest-arrival-time (a.k.a. shortest-path) route plans of private cars.
2. The application of all the necessary updates of the traffic metadata kept in the UTKB, in order to incorporate the online traffic as recorded by the emergency reports generated by the Crowd-sourcing Module.

During the pilot execution, several earliest-arrival-time queries were submitted to the TDR cloud-service. The following data were recorded:

- For each query, we recorded the origin location, the destination location, the departure time from origin, the arrival time to destination, the distance, and for a medium-sized diesel car, the fuel consumption and CO_2 equivalent emissions.
- For each emergency report, we recorded the emergency report id, the start point and end point of the affected road segment, the distance of the road segment, the old and new travel time traversing the road segment and the start and end time point as the duration of the new travel time update.
- For each query before and after the absorption of online traffic updates, we recorded the emergency report id, the origin location, the destination location, the departure time from origin, the arrival time to destination, the distance, and for a medium-sized diesel car, the fuel consumption and CO_2 equivalent emissions.

Our pilot consolidation results revealed that more than 70% of the users were very satisfied with the TDR service, while another 17% were satisfied. Most of the users found the TDR service useful and easy to use without encountering any technical problems.

Fig. 5 illustrates the procedure of an earliest-arrival-time query before and after an emergency report.

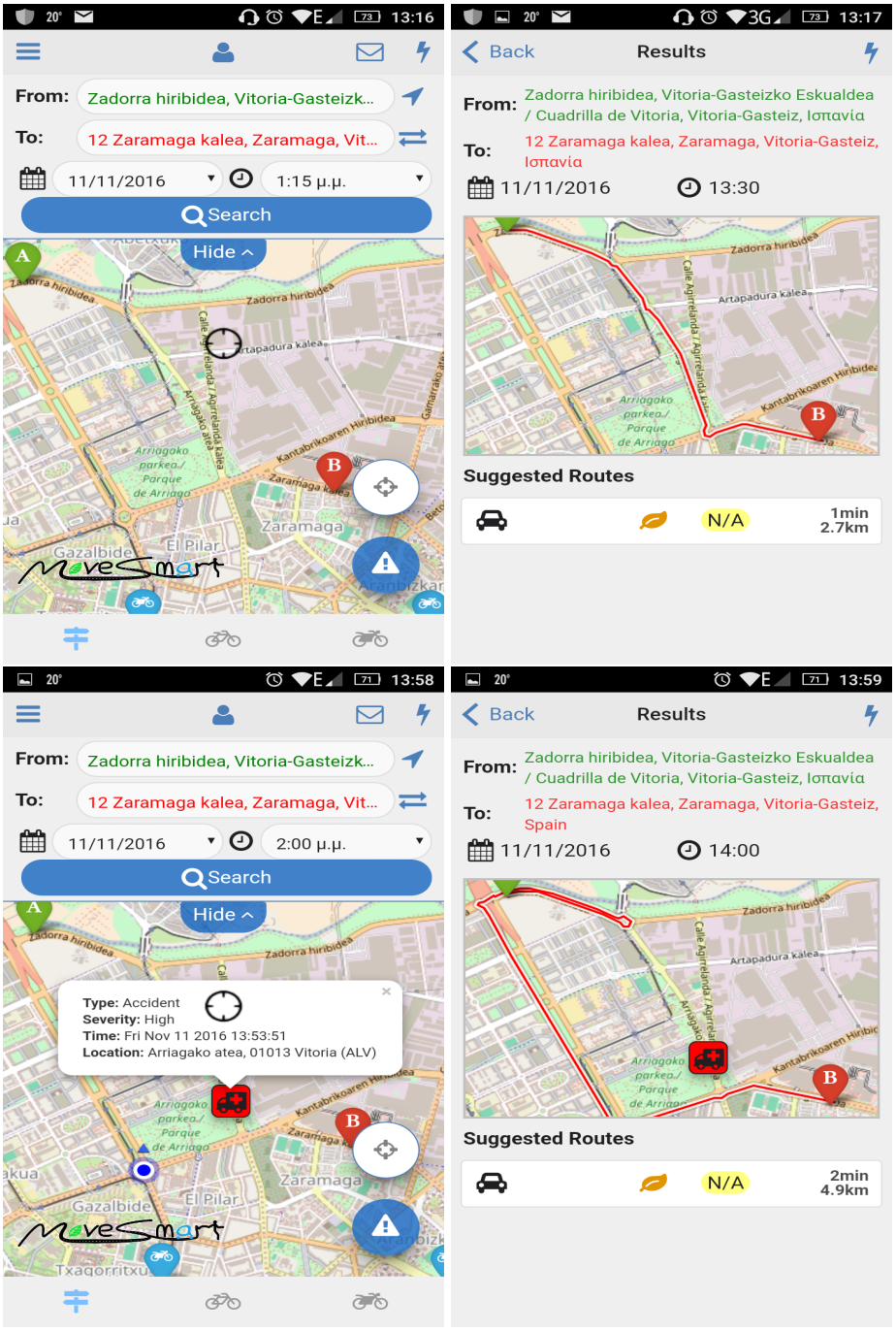

Fig. 5: Earliest-arrival-time query before and after an emergency report.

6 Conclusions

We presented the architecture of a cloud-based TDR service, aiming at providing fast real-time responses to arbitrary earliest-arrival-time queries as well as at updating efficiently the various traffic metadata kept in an urban traffic knowledge base, so that the service remains live-traffic aware. We also provided the implementation details of the core algorithmic TD routing engine, which is the most crucial module regarding the performance of the TDR service.

We plan to further enhance our cloud-based TDR service, in particular its core algorithmic TD engine, with the more sophisticated *hierarchical* algorithmic approaches presented recently in [6], which are expected to boost further the query time.

References

1. R. Agarwal and P. Godfrey. Distance oracles for stretch less than 2. In *24th Symp. on Discrete Algorithms (SODA'13)*, pp. 526–538. ACM-SIAM, 2013.
2. H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. Werneck. Route planning in transportation networks. *CoRR*, abs/1504.05140, 2015.
3. L. Foschini, J. Hershberger, and S. Suri. On the complexity of time-dependent shortest paths. *Algorithmica*, **68**(4):1075–1097 (2014)
4. KaHIP – Karlsruhe High Quality Partitioning, May 2014.
5. S. Kontogiannis, G. Michalopoulos, G. Papastavrou, A. Paraskevopoulos, D. Wagner, and C. Zaroliagis. Engineering oracles for time-dependent road networks. In *Algorithm Engineering and Experiments* – ALENEX 2016 (SIAM 2016), pp. 1-14.
6. S. Kontogiannis, D. Wagner, and C. Zaroliagis. Hierarchical Time-Dependent Oracles. In *Algorithms and Computation* – ISAAC 2016, LIPICs Vol. 64, pp.47:1-47:13. Also, CoRR `http://arxiv.org/abs/1502.05222` (2015)
7. S. Kontogiannis, and C. Zaroliagis. Distance oracles for time-dependent networks. *Algorithmica*, **74**(4):1404-1434 (2016).
8. MOVESMART project. `http://www.movesmartfp7.eu/`.
9. M. Patrascu and L. Roditty. Distance oracles beyond the Thorup-Zwick bound. *SIAM J. Comput.*, 43(1):300–311, 2014.
10. E. Porat and L. Roditty. Preprocess, set, query! *Algorithmica*, 67(4):516–528, 2013.
11. C. Sommer. Shortest-path queries in static networks. *ACM Computing Surveys*, 46, 2014.
12. C. Sommer, E. Verbin, and W. Yu. Distance oracles for sparse graphs. In *50th Symp. on Foundations of Computer Science (FOCS'09)*, pages 703–712, IEEE 2009.
13. M. Thorup and U. Zwick. Approximate distance oracles. *Journal of the ACM*, 52(1):1–24, 2005.
14. C. Wulff-Nilsen. Approximate distance oracles with improved preprocessing time. In *23rd Symp. on Discrete Algorithms (SODA'12)*, pp. 202–208, ACM–SIAM, 2012.
15. C. Wulff-Nilsen. Approximate distance oracles with improved query time. In *24th Symp. on Discrete Algorithms (SODA'13)*, pp. 539–549, ACM–SIAM 2013.

A Hueristics for Improving Performance

We describe some heuristic algorithmic improvements, as well as some implementation details that we apply during the creation and maintenance of traffic metadata in the UTKB, in order to obtain even better performance, both wrt the required preprocessing space and wrt the efficiency of the query phase.

Approximately constant functions. The TRAP approximation method introduces one intermediate breakpoint per interval that satisfies the required approximation guarantee. To keep our algorithm space-efficient, the first thing that we do, when dealing with every subinterval of the time period, is that we check for approximately constant functions. More precisely, we perform an additional sampling at the middle point t_m of each $[t_s, t_f)$ and we consider the upper-approximation $\overline{\Delta}[\ell, v][t_s, t_f)$ to be "almost constant", if the following condition holds: $D[\ell, v](t_s) = D[\ell, v](t_f) = D[\ell, v](t_m)$. For those approximately constant functions, the insertion of the additional breakpoint at t_m is unnecessary.

Piecewise composition. Many shortest paths are likely to contain at least one arc with piecewise travel time, making the shortest-path function also piecewise. In our case, keeping the predecessor vertex of v in every sample of $\overline{\Delta}[\ell, v](t)$, allows us to analyze any ℓ-v approximate shortest path into two sub-paths ℓ-p-v. Starting from the destination v, we travel back to a predecessor p, as long as all vertices u up to p have constantly the same predecessor kept in the corresponding samples of $\overline{\Delta}[\ell, u](t)$, and all arcs involved in the p-v subpath have a constant travel-time function. Therefore, there is no need to keep any samples for the approximate function $\overline{\Delta}[\ell, v](t)$; instead, we store the necessary information in the form: (*constant-travel-time*, *predecessor-p*, *approximate-path-predecessor*). In this way, the approximate travel-time function $\overline{\Delta}[\ell, v](t)$ is given by taking into account $\overline{\Delta}[p, v](t)$ and the (approximately) constant travel-time from v to p, i.e., $\overline{\Delta}[\ell, v](t) = \overline{\Delta}[\ell, p](t) + \overline{\Delta}[p, v](t)$. In our experiments, this method leads to around 40-50% reduction of the space requirements.

Delay shifts. In cases that such a predecessor p does not exist, we have to store the approximate travel-time function as a sequence of breakpoints. However, the samples collected for $\overline{\Delta}[\ell, v](t)$ may have small delay variation. Based on this fact, the required space can be further reduced. We store the minimum travel-time value and for each leg we only need to store the small shift from this value. This conversion leads to around 5-10% reduction of the required preprocessing space.

Fixed range. For a one-day time period, departure-times have a bounded value range. The same holds for travel-times which are at most one-day for any query within a country or city area. When the considered precision of the traffic data is within seconds, we handle time-values as integers in the range $[0, 86399]$, rather than real values, sacrificing precision for space reduction. In particular, we convert all floating-point time-values t_f to integers t_i with fewer bytes and a given

unit of measure. For a unit of measure s, the resulting integer is $t_i = \lceil t_f/s \rceil$ and needs size $\lceil \log_2(t_f/s)/8 \rceil$ bytes. Converting t_f to t_i results to an absolute error of at most 2 seconds. Therefore, for storing the time-values of approximate travel-time summaries, we can consider different resolutions, depending on the scale factor s, to achieve further reduction of the preprocessing space.

Compression. Since there is no need for all landmarks to be concurrently active, we can compress their data blocks. This method leads to significant reduction of the space requirements, especially for large-scale networks.

Contraction. The space of the preprocessed travel-time summaries can be further reduced if we consider a subset of vertices in the network as inactive. More precisely, we can conduct a preprocessing of the instance that contracts all vertices which are not junctions, i.e., they form paths with no intersections. Each such path can be represented by a single shortcut arc, which is added at the endpoints of the chain and equipped with an arc-traversal time function equal to the corresponding (exact) path-travel-time function. The arcs involved in the contracted paths are also considered as inactive. All contracted nodes are ignored and therefore the number of possible destinations from a landmark is smaller. At the query phase, these paths can be easily retrieved, by exploiting the appropriate information kept on all inserted shortcuts and all contracted nodes for this purpose.

Parallelism. We can speed up the preprocessing time for computing the one-to-all approximate travel-time functions, from properly selected landmarks towards all reachable destinations, as well as the real-time responsiveness to live-traffic reports, i.e., the re-computation of the travel-time summaries for the subset of affected landmarks, by exploiting the inherent parallelism of the entire process.